A Case Study in Using Formal Methods to Verify Programs That Use MPI One-Sided Communication

Salman Pervez [a], Ganesh Gopalakrishnan [a], Robert M. Kirby [a], Rajeev Thakur [b], and William Gropp [b]

[a] *School of Computing, University of Utah, Salt Lake City, UT 84112, USA*

[b] *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA*

Abstract

We used formal-verification methods based on model checking to analyze the correctness properties of one existing and two new distributed-locking algorithms implemented by using MPI's one-sided communication. Model checking exposed an overlooked correctness issue with the existing algorithm, which had been developed by relying solely on manual reasoning. Model checking also helped confirm the basic correctness properties of the two new algorithms. Performance evaluation of the new algorithms on up to 128 processors revealed that one of them outperforms the other, especially under heavy lock contention. Our experience is that MPI-based programming, especially the tricky and relatively poorly understood one-sided communication features, stand to gain immensely from model checking. This, as well as the existing practice of routinely employing model checking in many areas of concurrent hardware and software design, confirms that the MPI community can benefit greatly from the use of formal verification.

Key words: Message Passing Interface (MPI), one-sided communication, formal verification, model checking

Email addresses: `spervez@cs.utah.edu` (Salman Pervez), `ganesh@cs.utah.edu` (Ganesh Gopalakrishnan), `kirby@cs.utah.edu` (Robert M. Kirby), `thakur@mcs.anl.gov` (Rajeev Thakur), `gropp@mcs.anl.gov` (William Gropp).

1 Introduction

Concurrent protocols are notoriously hard to design and verify. Experience has shown that virtually all nontrivial protocol implementations contain bugs such as deadlocks, livelocks, and memory leaks, despite extensive care taken during design and testing. Most of these bugs are basic design errors due to "unexpected" (untested) concurrent behaviors. Therefore, it stands to reason that if finite-state models of these protocols are created and exhaustively analyzed for the desired formal properties, robust protocol implementations would result. The technology for such finite-state modeling, property description, and exhaustive analysis developed over the past three decades—known as *model checking* [3]—has been successfully applied to numerous software and hardware systems. There are many technology transfer success stories in this area, including the use of model checking in the Windows Device Driver Development Kit [2] and the fact that cache-coherence protocols in all modern processors are verified by using model checking [1]. Although the conceptual difficulties due to concurrency, as well as the prevalence of concurrent programming bugs in the area of parallel scientific programming, are equally serious, we find little evidence (other than the efforts cited below) of model checking being applied in this area.

In this paper, we report case studies that show the promise of model checking in the area of parallel scientific programming using MPI. In particular, we focus on algorithms employing one-sided communication [10]. Being (relatively) recently introduced and implemented, MPI one-sided communication is insufficiently understood and documented. One-sided communication involves shared-memory concurrency, which is known to be inherently harder to reason about than the message-passing concurrency of traditional MPI. Verification complexity is exacerbated by the weak ordering semantics of the "puts" and "gets" performed within a one-sided synchronization epoch. This paper demonstrates that, by using model checking, bugs in MPI programs that use one-sided communication can be caught easily, while expending only modest amounts of human and computer time.

After presenting background on MPI one-sided communication in Section 2, we provide an overview of model checking in Section 3. We then describe the design of an existing distributed byte-range locking algorithm [20] and its formal verification through model checking (Section 4). Model checking helped uncover the serious problem of a potential deadlock, which the authors of the algorithm were unaware of. Model checking also found a more benign problem of extra (zero-byte) sends in the algorithm, which might lend itself to an implementation-dependent correction using `MPI_Iprobe` and posted receives. However, this problem may well turn into a memory leak. We then present two other designs of the same algorithm, formally verify them using model checking, provide empirical observations to interpret these model-checking results, and study their performance on up to 128 processors on a Linux cluster (Section 5). In Section 6, we conclude with a discussion of future work.

To our knowledge, nobody has applied model checking to analyze programs that use MPI one-sided communication. Siegel et al. have used model checking to verify MPI programs that employ a limited set of two-sided MPI communication primitives [15–17]. We also have previously employed model checking to verify MPI programs that use two-sided constructs [11,12]. Kranzlmüller used a formal event-graph based method to help understand MPI program executions [7]. Matlin et al. used the SPIN model checker to verify parts of the MPD process manager used in MPICH2 [9].

2 MPI One-Sided Communication

For lack of space, we review only the features of MPI one-sided communication relevant to this paper. One feature in MPI one-sided communication, known as passive-target synchronization [10], allows processes to gain exclusive access to communication windows in a block of code bracketed by `MPI_Win_lock` and `MPI_Win_unlock` calls. Read and write accesses can be performed by a process holding exclusive access to a window through `MPI_Put` and `MPI_Get`. The main semantic difficulty stems from these put and get calls being not required to obey their syntactic program order in terms of when they are performed. It is well known (see, e.g., [18]) that such ordering guarantees are crucial to the correctness of even simple concurrent protocols such as Peterson's mutual exclusion. The specification of one-sided communication in MPI further exacerbates the issue by introducing a complex set of informally stated rules that can easily lead to contradictory interpretations. Common mistakes users make include nesting synchronization epochs on the same window object (such as a win_lock/unlock within a fence), doing read-modify-writes via a get-modify-put in the same synchronization epoch (even though gets and puts are defined to be nonblocking), and doing a put and a get to/from the same memory location in the same synchronization epoch. For example, the broadcast algorithms in Appendix B and C of [8] are incorrect because they rely on `MPI_Get` being a blocking function, which it is not. In implementations that take advantage of the nonblocking nature of `MPI_Get`, such as MPICH2 [19], the code will indeed go into an infinite loop. Since MPI one-sided communication can be implemented in a variety of ways [5], the result of making such mistakes is often implementation dependent: the program may work fine on some implementations and not on others.

3 Model Checking

Model checking consists of two steps: creating *models* of concurrent systems and traversing these models exhaustively, while checking the truth of desired properties. A model can be anything from a simple finite-state machine modeling the concurrent system to actual deployed code. Model checking is performed by creating—either manually or automatically—simplified models of the concurrent system to be verified, recording states and paths visited to avoid test repetitions (an extreme case of which is infinite looping), and checking the desired correctness properties, typically on the fly. Given that the size of the reachable state space of concurrent systems can be exponential in the number of concurrent processes, model checkers employ a significant number of algorithms as well as heuristics to achieve the effect of full coverage without ever storing entire state histories. The properties checked by a model checker can range from simple state properties such as *assert*s to complex temporal logic formulas that express classes of desired behaviors. Model checkers are well known for their ability to track down deadlocks, illegal states (*safety* [3] bugs) and starvation scenarios (*liveness* [3] bugs) that may survive years of intense testing. We consider *finite-state* model checking where the model of the concurrent system is expressed in a modeling language—Promela [6] in our case (all the pseudocodes expressed in this paper have an almost direct Promela encoding once the MPI constructs have been accurately modeled). By (in effect) exhaustively traversing the concurrent-system automaton, a model checker helps establish desired temporal properties, such as "always P" and "A implies eventually Q," or generates concrete error traces when such properties fail.

The capabilities of model checkers have been steadily advancing, with modern model checkers being able to handle astronomically large (e.g., billions of) state spaces. A model checker that has received wide attention among the computer science community is SPIN [6]. Despite the very large state spaces of the SPIN MPI models discussed in this paper, our model-checking runs finished within acceptable durations (often in minutes) on standard workstations.

4 Formal Verification of Byte-Range Locking

Often, processes must acquire exclusive access to a range of bytes, such as a portion of a file. In [20], Thakur et al. presented an algorithm by which processes can coordinate among themselves to acquire byte-range locks, without a central lock-granting server. The algorithm uses MPI one-sided communication with passive-target synchronization (`MPI_Win_lock` and `MPI_Win_unlock`). We first describe the algorithm briefly, followed by a description of how we model checked it. Because of space limits, we cannot present the full pseudocode of the original algorithm; the reader may refer to the original paper [20] for details.

4.1 The Byte-Range Locking Algorithm

Each process keeps in a single common memory window (`lockwin`) its state consisting of a `flag` (initialized to 0) and the `start` and `end` values for the byte range (initialized to -1). A flag of 0 indicates that the process does not have the lock, while 1 indicates that it either has acquired the lock or wants to acquire the lock. A process updates its state and reads others' states by acquiring exclusive access to `lockwin` and making `MPI_Put` and `MPI_Get` calls. Since the processes acquire exclusive access, the actions of any one process on `lockwin` are guaranteed to be atomic with respect to the actions of other processes.

In order to acquire the lock, a process sets its `flag` to 1, updates its `start` and `end` values, and gets the corresponding values of other processes. It then checks whether any other process has set a conflicting byte range and has a flag value of 1. If it does not find such a process, it assumes that it has acquired the lock. Otherwise, it assumes that it does not have the lock, resets its flag to 0 via another lock-put-unlock, and blocks on a zero-byte `MPI_Recv` call, waiting for a process that has the lock to wake it up with a zero-byte send. The process will retry the lock after receiving the message. To release a lock, a process again acquires exclusive access, resets its flag to 0 and its start and end offsets to -1, and gets the values of other processes. If it finds a process with a conflicting byte-range (ignoring the flag), it sends a zero-byte message (via `MPI_SEND`) to wake up that process.

4.2 Checking the Byte-Range Locking Algorithm

To model the algorithm, we first needed to model the MPI one-sided communication constructs used in the algorithm and capture their semantics precisely as specified in the MPI Standard [10]. For example, the MPI Standard specifies that if a communication epoch is started with `MPI_Win_lock`, it must end with `MPI_Win_unlock` and that the put/get/accumulate calls made within this epoch are not guaranteed to complete before `MPI_Win_unlock` returns. Furthermore, there are no ordering guarantees of the puts/gets/accumulates within an epoch. Therefore, in order to obtain adequate execution-space coverage, *all permutations of put/get/accumulate calls in the epoch must be examined.* However, the byte-range locking algorithm uses the `MPI_LOCK_EXCLUSIVE` lock type, which means that while a certain process has entered the synchronization epoch, no other process may enter until that process has left. This makes the synchronization epoch an atomic block and renders all permutations of the calls within it equivalent from the perspective of other processes. Modeling the byte-range locking algorithm itself was relatively straightforward. (This experience augurs well for the checking of other algorithms that use MPI one-sided communication, as one of the significant challenges in model checking lies in the ease of modeling constructs in the target domain using modeling primitives in the modeling language.) The complete Promela code used in our model checking can be found online [13].

When we model checked our model with three processes, our model checker, SPIN [6], discovered an error indicating an "invalid end state." Deeper probing revealed the following error scenario (explained through an example, which assumes that P1 tries to lock byte-range $\langle 1, 2\rangle$, P2 tries to lock $\langle 3, 4\rangle$, and P3 tries to lock $\langle 2, 3\rangle$):

- P1 and P3 successfully acquire their byte-range locks.
- P2 then tries to acquire its lock, notices conflict with respect to both P1 and P3, and blocks on the `MPI_Recv`.
- P1 and P3 release their locks, both notice conflicts with P2, and both perform an MPI_Send, when only one send is needed.

Hence, while P2 ends up successfully waking up and acquiring the lock, the extra `MPI_Send`s may accumulate in the system. This is a subtle error whose severity depends on the MPI implementation being used. Recall that the MPI Standard allows implementors to decide whether to block on an `MPI_Send` call. In practice, a zero-byte send will rarely block. Nonetheless, an implementation of the byte-range locking algorithm can address this problem by periodically calling `MPI_Iprobe` and matching any unexpected messages with `MPI_Recv`s.

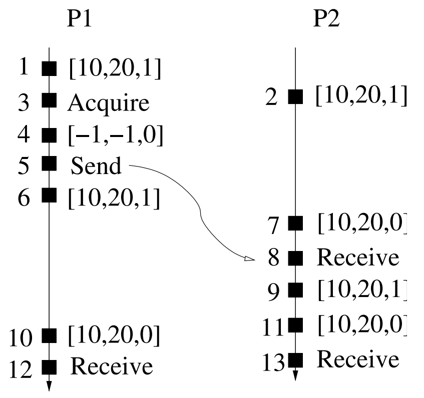

Fig. 1. A deadlock scenario found through model checking.

We then modeled the system as if these extra `MPI_Send`s do not exhaust the system resources and hence do not cause processes to block. In this case, model checking detected a far more serious deadlock situation, summarized in Figure 1. P1 expresses its intent to acquire a lock in the range $\langle 10, 20\rangle$ (1), with P2 following suit (2). P1 acquires the lock (3), finishes using it and relinquishes it (4), and performs a send to unblock P2 (5). Before P2 gets a chance to change its global state, P1 tries to reacquire the lock (6). P1 reads P2's current flag value as 1, so it decides to block by carrying out events (10) and (12). At this point, P2 changes its global state, receives the message sent by P1 (8), and proceeds to reacquire the lock (9). P2 reads P1's current flag value as 1, so it decides to block by carrying out events (11) and (13). Both processes now block on receive calls, and the result is deadlock. We note that the authors of the algorithm were unaware of this problem until the model checker found it!

5 Correcting the Byte-Range Locking Algorithm

We propose two approaches to fixing this deadlock problem, describe our experience with using model checking on these solutions, and study their relative performance on a Linux cluster.

5.1 Alternative 1

One way to eliminate deadlocks is to add a third state to the "flag" used in the algorithm. This is shown in the pseudocode in Figure 2. In the original algorithm, a flag value of '0' indicates that the process does not have the lock, while a flag value of '1' indicates that it either has acquired the lock or is in the process of determining whether it has acquired the lock. In other words, the '1' state is overloaded. In the proposed fix, we add a third state of '2' with '0' denoting the same as before, '1' now denoting that the process has acquired the lock, and '2' denoting that it is in the process of determining whether it has acquired the lock. There is no change to the lock-release algorithm, but the lock-acquire algorithm changes as follows.

When a process wants to acquire a lock, it writes its flag value as '2' and its start and end values in the memory window. It also reads the state of the other processes from the memory window. If it finds a process with a conflicting byte range and a flag value of '1', it knows that it does not have the lock. So it resets its flag value to '0' and blocks on an `MPI_Recv`. If no such process (with conflicting byte range and flag=1) is found, but there is another process with a conflicting byte range and a flag value of '2,' the process resets its flag to '0,' its start and end offsets to -1, and retries the lock from scratch. If neither of these cases is true, the process sets its flag value to '1' and considers the lock acquired.

Dealing with Fairness and Livelock. One problem with this algorithm is the issue of fairness: a process wanting to acquire the lock repeatedly may starve out other processes. This problem can be avoided if the MPI implementation grants exclusive access to the window fairly among the requesting processes.

```
Lock_acquire (int start, int end)
{
  val[0] = 2; /* flag */
  val[1] = start; val[2] = end;
  /* add self to locklist */
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
  MPI_Put(&val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
  MPI_Get(locklistcopy, 3*(nprocs-1), MPI_INT, homerank, 0, 1,
          locktype1, lockwin);
  MPI_Win_unlock(homerank, lockwin);
  /* check to see if lock is already held */
  flag1 = flag2 = 0;
  for (i=0; i < (nprocs-1); i ++) {
    if ((flag == 1) && (byte ranges conflict with lock request)) {
      flag1 = 1;
      break;
    }
    if ((flag == 2) && (byte ranges conflict with lock request)) {
      flag2 = 1;
      break;
    }
  }
  if (flag1 == 1) {
    /* reset flag to 0, wait for notification, and then retry */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 0;
    MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* wait for notification from some other process */
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm, &status);
    /* retry the lock */
    Lock_acquire(start, end);
  }
  else if (flag2 == 1) {
    /* reset flag to 0, start/end offsets to -1 */
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 0; /* flag */
    val[1] = -1; val[2] = -1;
    MPI_Put(val, 3, MPI_INT, homerank, 3*myrank, 3, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* wait for small random amount of time (to avoid livelock) */
    /* then retry the lock */
    Lock_acquire(start, end);
  }
  else {
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
    val[0] = 1;
    MPI_Put(val, 1, MPI_INT, homerank, 3*myrank, 1, MPI_INT, lockwin);
    MPI_Win_unlock(homerank, lockwin);
    /* lock is acquired */
  }
}
```

Fig. 2. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 1).

Even in the absence of this problem, model checking revealed the potential for livelock in one particular situation when processes try to acquire the lock multiple times. The situation is similar to the one that caused deadlock in the original algorithm, where two processes in the state of trying to acquire the lock both block in order that the other can go ahead. To avoid deadlock, we introduced the intermediate state of 2, which ensures that instead of blocking

P1 sets flag=2
P2 sets flag=2, sees P1's 2, and decides to retry
P1 acquires the lock
P1 releases the lock and retries for the lock
P1 sets flag=2, sees P2's 2, and decides to retry

P2 sets flag=0
P2 sets flag=2, sees P1's 2, and decides to retry
P1 sets flag=0
P1 sets flag=2, sees P2's 2, and decides to retry

Above sequence repeats

Fig. 3. A potential livelock scenario found through model checking.

on an `MPI_Recv`, the process backs off and retries the lock. Figure 3 shows an example of how the backoff and retry could repeat forever if events get scheduled in a particular way. Note that it is very hard for humans to detect the possibility of such a livelock just by studying the algorithm. We ourselves were unaware of it until the model checker found it.

The livelock can be prevented by having each process backoff for a random amount of time before retrying, thereby avoiding the likelihood of the same sequence of events occurring each time.

Alternative 2 presented below avoids the possibility of livelock.

5.2 Alternative 2

This approach uses the same values for the flag as the original algorithm; but when a process tries to acquire a lock and determines that it does not have the lock, it picks a process (that currently has the lock) to wake it up and then blocks on the receive. For this purpose, we add a fourth field (the pick field) to the values for each process in the memory window (see Figure 4). The process about to block must now decide whether to block. This decision is based on two factors: (i) Has the process selected to wake it up already released the lock? and (ii) Is there a possibility of a deadlock caused by a cycle of processes that wait on each other to wake them up? The latter can be detected and avoided by using the algorithm in Figure 5. The former can be easily determined by reading the values returned by the `MPI_Get` on line 26. If the selected process has already released the lock, a new process must be picked in its place. We simply traverse the list of conflicting processes until we find one that has not yet released the lock. If no such process is found, the algorithm tries to reacquire the lock. Note the added complexity of going through the list of conflicting processes and doing put and get operations each

```
Lock_acquire (int start, int end)
{
  int picklist[num_procs];
  val[0] = 1; /* flag */
  val[1] = start; val[2] = end; val[3] = -1; /* pick */
  /* add self to locklist */
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
  MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
  MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
          locktype1, lockwin);
  MPI_Win_unlock(homerank, lockwin);
  /* check to see if lock is already held */
  cprocs_i = 0;
  for (i=0; i <(nprocs-1); i ++)
    if ((flag == 1) && (byte range conflicts with Pi's request)) {
      conflict = 1; picklist[cprocs_i] = Pi; cprocs_i++;
    }
  if (conflict == 1) {
    for (j=0; j <cprocs_i; j++) {
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, homerank, 0, lockwin);
      val[0] = 0; val[3] = picklist[j];
      /* reset flag to 0, indicate pick and pick_counter */
      MPI_Put(&val, 4, MPI_INT, homerank, 4*myrank, 4, MPI_INT, lockwin);
      MPI_Get(locklistcopy, 4*(nprocs-1), MPI_INT, homerank, 0, 1,
              locktype1, lockwin);
      MPI_Win_unlock(homerank, lockwin);
      if (picklist[j] has released the lock || detect_deadlock())
        /* repeat for the next process in picklist */
        j++;
      else {
        /* wait for notification from picklist[j], then retry the lock */
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, WAKEUP, comm,
                 MPI_STATUS_IGNORE);
        break;
      }
    }
    Lock_acquire(start, end);
  }
  /* lock is acquired */
}
```

Fig. 4. Pseudocode for the deadlock-free byte-range locking algorithm (Alternative 2).

```
detect_deadlock() {
  cur_pick = locklistcopy[4 * myrank + 3];
  while(i < num_procs) {
    /* picking this process means a cycle is completed */
    if(locklistcopy[4 * cur_pick + 3] == my_rank) return 1;
    /* no cycle can be formed  */
    else if(locklistcopy[4 * cur_pick + 3] == -1) return 0;
    else cur_pick = locklistcopy[4 * cur_pick + 3];
  }
}
```

Fig. 5. Avoiding circular loops among processes picked to wake up other processes in Alternative 2.

time. However, if this loop is successful and the process blocks on `MPI_Recv`, we can save considerable processor time in the case of highly contentious lock requests as compared with Alternative 1.

```
inline lock_acquire(start, end) {
  .......
      do
      :: 1 ->
         do
           :: cprocs_i == NUM_PROCS ->
                break;
           :: else ->
                cprocs[cprocs_i] = -1;
                cprocs_i++;
         od;
         cprocs_i = 0;

         MPI_Win_lock(my_pid);
         MPI_Put(1, start, end, -1, my_pid);
         MPI_Get(my_pid);
         MPI_Win_unlock(my_pid);

         do
           :: lock_acquire_i == DATA_SIZE -> break;
           :: else ->
                if
                  :: lock_acquire_i == my_pid ->
                            lock_acquire_i = lock_acquire_i + 4;
                  :: lock_acquire_i != my_pid &&
                     (private_data[my_pid].data[lock_acquire_i] == 1) ->
                          /* someone else has a flag value of 1,
                          check for byte range conflict */

              ...rest of the code omitted...
```

Fig. 6. Excerpts from Promela encoding of `Lock_acquire`.

5.3 Formal Modeling and Verification

Both alternative algorithms were prototyped by using Promela. The entire code is available online [13]. We employ Promela channels to model MPI sockets. While the creation of such Promela models requires some expertise, our experience is that Promela can be easily taught to most engineers. Our Promela models occupy nearly the same number of lines of code and are structured similar to the pseudocode we have presented for the algorithms. Figure 6 shows an excerpt of the `lock_acquire` function in Promela. Comparing with Figure 4, we see that the Promela code fleshes out the pseudocode by adding an iteration across `NUM_PROCS` and essentially carries out the same sequence of actions such as `MPI_Win_lock` and `MPI_Put`. We have built a support library in Promela that models these MPI primitives.

5.4 Assessment of the Alternative Algorithms

We model checked these algorithms using SPIN, which helped establish the following formal properties of these algorithms:

- Absence of deadlocks (both alternatives).

- Communal progress (that is, if a collection of processes request a lock, then someone will eventually obtain it). Alternative 2 satisfies this under all fair schedules (all processes are scheduled to run infinitely often), whereas Alternative 1 places a few additional restrictions to rule out a few rare schedules (the livelock problem) [14].

We note that neither of these alternatives eliminates the extra sends, but, as described in Section 4, an implementation can deal with them by using `MPI_Iprobe`. That said, Alternative 2 considerably reduces these extra sends, as it restricts the number of processes that can wake up a particular process compared with Alternative 1. We are still seeking algorithms that would avoid the extra sends (and be efficient).

5.5 Performance Results

To measure the relative performance of the two algorithms, we wrote three test programs: one in which all processes try to acquire nonconflicting locks (different byte ranges), another in which all processes try to acquire a conflicting lock (same byte range), and a third in which all processes acquire random locks (random byte range between 0 and 1000). In all tests, each process acquires and releases the lock in a loop several times. We measured the time taken by all processes to complete acquiring and releasing all their locks and divided this time by the number of processes times the number of iterations. This measurement gave the average time taken by a single process for acquiring and releasing a single lock. We ran the tests on up to 128 nodes of a Myrinet-connected Linux cluster at Argonne. We used the latest version of MPICH2 (1.0.5) with the Nemesis channel over GM.

Figure 7 shows the results for nonconflicting locks. In this case, there is no contention for the byte-range lock; however, since the target window is located on one process (rank 0) and all the one-sided operations are directed to it, the time taken to service the one-sided operations increases as the number of processes calling them increases. Alternative 1 is slightly slower than Alternative 2 because it always involves two steps: the flag is first set to 2, and if no conflict is detected, it is set to 1. Alternative 2 requires only one step.

Figure 8 shows the results for conflicting and random locks. Even in these two cases, we find that Alternative 2 outperforms Alternative 1. Alternative 1 is hampered by the need for a process to back off for a random amount of time before retrying the lock when it detects that another process is trying to acquire a lock (flag=2). This random wait is needed to avoid the livelock condition described earlier. We implemented the wait by using the POSIX `nanosleep` function, which delays the execution of the program for at least the

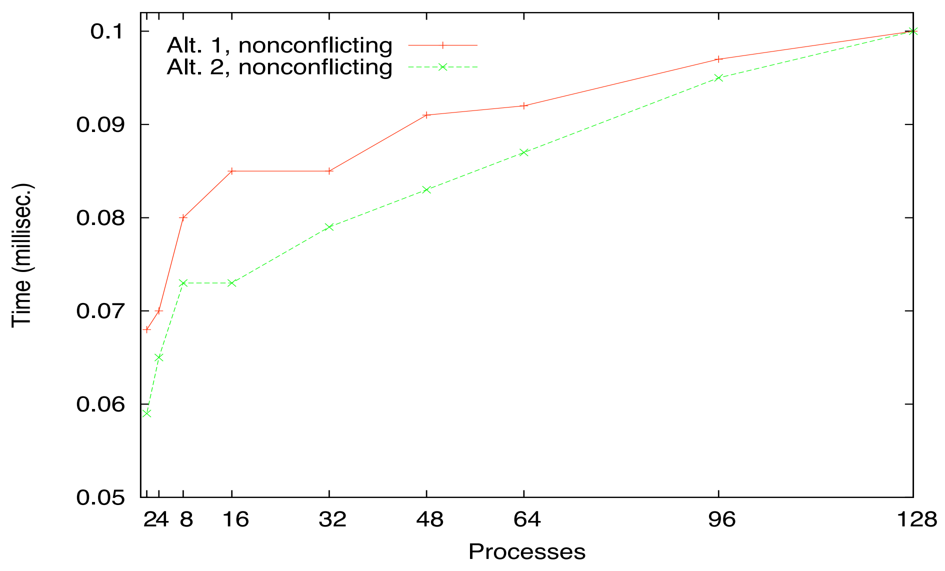

Fig. 7. Average time per process for acquiring and releasing a nonconflicting lock.

time specified. However, a drawback of the way this function is implemented in Linux (and many other operating systems) is that even though the specified time is small, it can take up to 10 ms longer than specified until the process becomes runnable again. This causes the process to wait longer than needed and slows the algorithm. Also, there is no good way to know how long a process must wait before retrying in order to avoid the livelock. Based on experiments, we used a time equal to (myrank*500) nanoseconds, where myrank is the rank of the process.

For conflicting locks on a small number of processes (4–16), we find that the time taken by Alternative 1 is substantially higher than Alternative 2. We believe this is because the effect of `nanosleep` taking longer than specified to return is more visible here as wasted time. On larger numbers of processes, that time gets used by some other process trying to acquire the lock and hence does not adversely affect the average.

6 Conclusions and Future Work

We have shown how formal verification based on model checking can be used to find actual deadlocks in published algorithms that use the MPI one-sided communication primitives. We have also discussed how this technology can help shed light on a number of related issues such as forward progress and the possibility of there being unconsumed messages. We presented and analyzed two deadlock-free algorithms for byte-range locking and verified their characteristics.

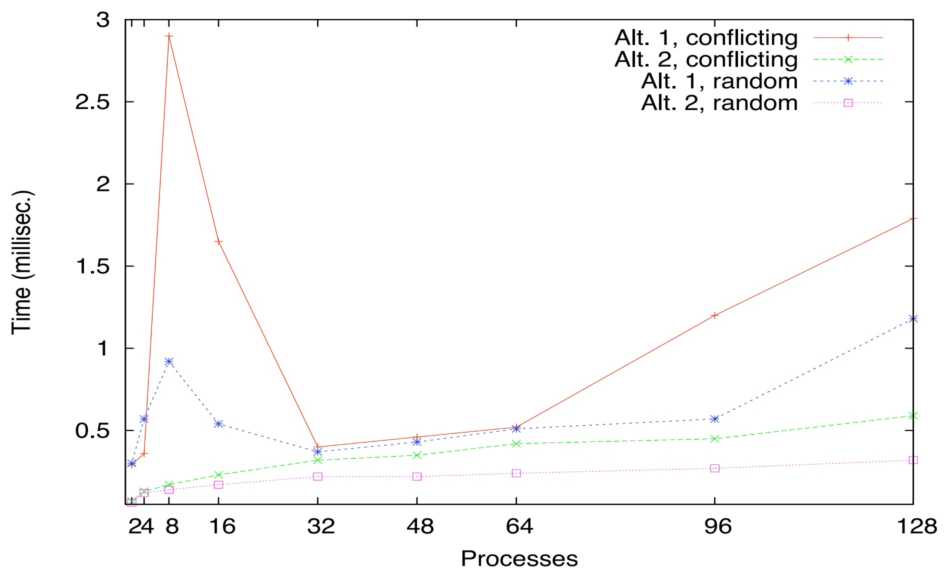

Fig. 8. Average time per process for acquiring and releasing a conflicting lock and a random lock. The spike in the small process range for Alternative 1 is because the effect of the backoff with nanosleep is more visible here as wasted time, whereas on larger number of processes, that time gets used by some other process trying to acquire the lock.

Nonetheless, our work in this field is still in its early stages. Capitalizing on the maxim that formal methods can have their biggest impact when applied to constructs that are relatively new or are under development, we plan to formalize the entire set of MPI one-sided communication primitives. This can help develop a comprehensive approach to verifying programs that use the MPI one-sided constructs. As future case studies, we will analyze other algorithms, such as the scalable fetch-and-increment algorithm described in [4]. We plan to explore the use of automated tools to extract models from MPI programs, instead of creating them by hand.

We are also working on an in situ approach to model checking MPI programs directly at execution time, without first creating a separate model for the program. This approach helps avoid the modeling tedium, as well as the possibility of introducing errors in the modeling itself. During in situ model checking, MPI process executions are controlled by an explicit scheduler process. Although the scheduler is in the early stages of development, it can detect deadlocks in MPI source programs without any modeling whatsoever. It has successfully caught the deadlock bug described in this paper by systematically exploring interleavings of the corresponding MPI program. Making in situ model checking efficient as well as portable across various MPI library implementations will permit us to place the model-checking technology in the hands of even more practitioners.

Acknowledgments

This work was supported by NSF award CNS-0509379, by the Microsoft HPC Institutes program, and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

[1] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[2] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of IFM 04: Integrated Formal Methods*, pages 1–20. Springer, April 2004.

[3] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[5] William Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 415–424. LNCS 3666, Springer, September 2005.

[6] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, MA, 2003.

[7] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, John Kepler University Linz, Austria, September 2000. `http://www.gup.uni-linz.ac.at/~dk/thesis`.

[8] Glenn R. Luecke, Silvia Spanoyannis, and Marina Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.

[9] Olga Shumsky Matlin, Ewing Lusk, and William McCune. SPINning parallel systems software. In *Model Checking of Software: 9th International SPIN Workshop*, pages 213–220. LNCS 2318, Springer, 2002.

[10] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[11] Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *Workshop on Software Model Checking*, 2005. Electronic Notes on Theoretical Computer Science (ENTCS), No. 953.

[12] Robert Palmer, Ganesh Gopalakrishnan, and Mike Kirby. Formal specification and verification using +CAL: An experience report. In *Proceedings of Verify'06 (FLoC 2006)*, 2006.

[13] Salman Pervez, 2006. Promela Encoding of Byte Range Locking Written Using MPI One-sided Operations, at `http://www.cs.utah.edu/formal_verification/europvm06-promela-code`.

[14] Salman Pervez. Byte-range locks using MPI one-sided communication. Technical report, University of Utah, School of Computing, 2006. `http://www.cs.utah.edu/formal_verification/OnesidedTR1/`.

[15] Stephen F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 413–429, 2005.

[16] Stephen F. Siegel and George S. Avrunin. Verification of MPI-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, pages 286–303. LNCS 2989, Springer, April 2004.

[17] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, July 2006.

[18] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 2001.

[19] Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in MPI one-sided communication. *International Journal of High-Performance Computing Applications*, 19(2):119–128, Summer 2005.

[20] Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 120–129. LNCS 3666, Springer, September 2005.